

Typed Transformations of Typed Abstract Syntax Presentation

Bas van Gijzel

April 7, 2010

1 The problem: optimising an EDSL

EDSL = **E**mbbeded **D**omain **S**pecific **L**anguage.

Good example: *Parser combinators*.

```
pMany p = (:) <$> p <*> pMany p <|> pReturn [] $
```

An EDSL is still Haskell. Combinators of the EDSL are actually represented by Haskell functions.

Assume we want to optimise?

- You can hope GHC solves all.
- Try global analysis or transformations on the functions.
 - Inspecting functions?
 - Equality of functions?

Apparently that does **not** work.

The (partial) solution: Represent EDSL by an **A**bstract **S**yntax **T**ree (**AST**).

However, an EDSL is typed by the underlying language. We would lose information or possibly make the program type incorrect by transforming the syntax tree.

Full solution: Represent EDSL by a **typed** AST.

An optimisation must keep the types, therefore we now have:

Typed Transformations of Typed Abstract Syntax (see also the paper by Baars and Swierstra [1])

2 Outline

- I will introduce increasingly complex expression EDSLs.
- These EDSLs will cover bindings/variables.
- The bindings will need an environment to keep track of.
- One of the key achievements of the TTTAS paper is the manipulation of such environments.
- This general manipulation of environments make it suitable for multiple applications
- I hope to make a decent start at introducing the library to do so.
- Application: a very brief view on the CHRISTMASTREE package.

3 A simple expression EDSL by using GADTs

(A compilable .lhs file)

```
{-# LANGUAGE GADTs, KindSignatures, ExistentialQuantification, RankNTypes #-}  
module TTTAS where  
import Prelude hiding (lookup, exp)
```

Consider an example similar to the AFP lecture on GADTS. We want to write a nice expression EDSL.

3.1 Expression EDSL: Attempt 1 (the basic language)

```
data Ex =  
  IntVal' Int  
  | BoolVal' Bool  
  | Add' Ex Ex  
  | LessThan' Ex Ex  
  | If' Ex Ex Ex  
deriving Show
```

Here the imagined concrete syntax for the EDSL would look like:

```
if 3 + 1 < 4 then 5 else 1 + 2
```

This would then be represented as an AST by:

```
exp1 :: Ex  
exp1 = If' (LessThan' (Add' (IntVal' 3) (IntVal' 1)) (IntVal' 4))  
          (IntVal' 5)  
          (Add' (IntVal' 1) (IntVal' 2))
```

We now have a nice EDSL, so the next step would be an interpreter:

```

data Val =
  VInt Int
  | VBool Bool
  deriving Show
eval1 :: Exp → Val
eval1 (IntVal'    n) = VInt n
eval1 (BoolVal'   b) = VBool b
eval1 (Add'       e1 e2) = case (eval1 e1, eval1 e2) of
  (VInt n1, VInt n2) → VInt (n1 + n2)
  -                  → error "type error"
eval1 (LessThan' e1 e2) = case (eval1 e1, eval1 e2) of
  (VInt n1, VInt n2) → VBool (n1 < n2)
  -                  → error "type error"
eval1 (If' e1     e2 e3) = case eval1 e1 of
  VBool b → if b then eval1 e2 else eval1 e3
  -      → error "type error"

```

And as expected:

```

*TTTAS> eval1 exp1
VInt 3

```

However:

- We have to throw type errors everywhere.
- The if case can have two different types.
- This does not scale nicely to a larger language.

3.2 Expression EDSL: Attempt 2 (typing expressions)

So here's a second attempt using GADTs. The expression datatype *Exp* is parametrised with the type of the expression.

```

data Exp    :: * → * where
  IntVal''   :: Int           → Exp Int
  BoolVal''  :: Bool          → Exp Bool
  Add''      :: Exp Int → Exp Int → Exp Int
  LessThan'' :: Exp Int → Exp Int → Exp Bool
  If''       :: Exp Bool → Exp a → Exp a → Exp a

```

And again an evaluator:

```

-- We don't need an extra datatype for values, just type a
eval2 :: Exp a → a
eval2 (IntVal''    n) = n
eval2 (BoolVal''   b) = b
eval2 (Add''       e1 e2) = eval2 e1 + eval2 e2
eval2 (LessThan'' e1 e2) = eval2 e1 < eval2 e2
eval2 (If''        e1 e2 e3) = if eval2 e1 then eval2 e2 else eval2 e3

```

And as expected:

```
exp2 :: Exp Int -- only type possible!  
exp2 = If'' (LessThan'' (Add'' (IntVal'' 3) (IntVal'' 1)) (IntVal'' 4))  
          (IntVal'' 5)  
          (Add'' (IntVal'' 1) (IntVal'' 2))
```

```
*TTTAS> eval2 exp2
```

```
3
```

Improvements:

- We don't crash at runtime by throwing type errors.
- We don't need tags for the final values.

However this is still not a very realistic/useful example. So we extend the expression EDSL by adding bindings/variables.

One thing that makes this hard: the references also need to be typed!

3.3 Expression EDSL: Attempt 3 (adding bindings)

So here's a third attempt, again using GADTs.

The expression datatype *Expr* is parametrised with an additional type for the environment containing the bindings.

The environment is represented as nested products (tuples within tuples).

This is very similar to the typed stack EDSL from the AFP exercises!

```
data Expr :: * -> * -> * where  
  Var      :: Ref a env          -> Expr a env  
  IntVal   :: Int               -> Expr Int env  
  BoolVal  :: Bool              -> Expr Bool env  
  Add      :: Expr Int env -> Expr Int env -> Expr Int env  
  LessThan :: Expr Int env -> Expr Int env -> Expr Bool env  
  If       :: Expr Bool env -> Expr a env  
                    -> Expr a env -> Expr a env
```

The *Ref a env* datatype is kept secret for now.

3.3.1 The incomplete evaluator

Again we define an evaluator with *lookup* left unspecified for now:

```
lookup :: Ref a env -> env -> a
```

```
eval3 :: Expr a env -> env -> a  
eval3 (Var      r) e = lookup r e  
eval3 (IntVal   n) _ = n  
eval3 (BoolVal  b) _ = b
```

```

eval3 (Add      e1 e2) e = eval3 e1 e + eval3 e2 e
eval3 (LessThan e1 e2) e = eval3 e1 e < eval3 e2 e
eval3 (If e1     e2 e3) e = if eval3 e1 e then eval3 e2 e else eval3 e3 e

```

This will work fine, but how do we define *lookup* and *Ref a env*?

Before diving into their definitions we need some more motivation regarding type level references.

So first an introduction to equality types using GADTs.

3.3.2 Equality of types using GADTs

Encoding of type equality is quite hard, using GADTs makes the problem a lot easier.

```

data Equal :: * -> * -> * where
  Eq :: Equal a a

```

Properties such as reflexivity, symmetry and transitivity of equality are also easily encoded:

```

reflex    :: Equal a a
reflex    = Eq
symm     :: Equal a b -> Equal b a
symm Eq = Eq
trans    :: Equal a b -> Equal b c -> Equal a c
trans Eq Eq = Eq

```

The important lesson here is that we can encode a lot of information about types using types!

3.3.3 The complete evaluator using typed references

We make references unique by using Peano numbering. (Alternative numbering is possible.)

```

data Ref a env where
  Zero :: Ref a (a, env')
  Suc  :: Ref a env' -> Ref a (x, env')

-- some example unique typed references
type ExampleEnv = (Int, (Char, (Bool, ())))
example :: ExampleEnv
example = (1, ('a', (True, ())))
refone  :: Ref Int ExampleEnv
refone  = Zero
refa    :: Ref Char ExampleEnv
refa    = Suc Zero
refb    :: Ref Bool ExampleEnv
refb    = Suc (Suc Zero)

```

Now that we have a unique numbering and typed references we can define equality on references.

```
-- a nice example of using the type of the references and the equality type
match :: Ref a env → Ref b env → Maybe (Equal a b)
match Zero Zero = Just Eq
match (Suc x) (Suc y) = match x y
match _ _ = Nothing
```

Considering this example usage of type level information defining *lookup* or even *update* is not that hard anymore:

```
lookup :: Ref a env → env → a
lookup Zero (a, _) = a
lookup (Suc r) (_, e) = lookup r e
update :: (a → a) → Ref a env → env → env
update f Zero (a, e) = (f a, e)
update f (Suc r) (x, e) = (x, update f r e)
```

Let's try some example runs with example environments:

```
type ExampleEnv = (Int, (Char, (Bool, ())))
example :: ExampleEnv
example = (1, ('a', (True, ())))
refone  :: Ref Int ExampleEnv
refone  = Zero
refa    :: Ref Char ExampleEnv
refa    = Suc Zero
refb    :: Ref Bool ExampleEnv
refb    = Suc (Suc Zero)
```

```
*TTTAS> lookup refa example
'a'
*TTTAS> lookup refb example
True
*TTTAS> lookup refone example
1
```

So now we can finally test some expressions:

```
varone = Var refone
varb   = Var refb
testexpr :: Expr Int ExampleEnv
testexpr = If varb (IntVal 3) varone
```

And finally for the example run:

```
*TTTAS> eval3 testexpr example
3
```

So now we've nearly defined a full-fledged expression EDSL and therefore also a pretty good corresponding environment.

However we still have one difficult feature to add: **mutual recursive terms**.

3.4 Expression EDSL: Final version (mutual recursion)

Examples of mutual recursive terms:

```
a = 1 : b
b = 2 : a
(y, x) = (5 + x, y)
```

However we can't represent this with the current types of *Ref a env* and *Expr a env*.

The current *env* only contains values and not terms. When we would use the current *env* parameter we would need to refer to env itself in env and get infinite type recursion (infinite combinations of tuples), which is not allowed in Haskell.

Thus we introduce a data type for the env parameter which solves the problem by splitting the type parameter in two parameters.

t is the type of the terms, for us that is *Expr*, *use* indicating the environment addressed by the references occurring in terms and *def* is constructed by the sequence of terms/definitions.

We again have a tupling of references but also a consing of usages.

```
data Env t use def where
  Empty :: Env t use ()
  Cons :: t a use → Env t use def' → Env t use (a, def')
```

The recursive data definitions can now be defined as follows:

```
type FinalEnv t usedef = Env t usedef usedef
type ExampleEnv2 = (Int, (Int, ()))
x  :: Expr Int ExampleEnv2
x  = Var Zero
y  :: Expr Int ExampleEnv2
y  = Var (Suc Zero)
rhss :: FinalEnv Expr ExampleEnv2
rhss = Cons (Add (IntVal 5) x) (Cons y Empty)
```

The *lookup* and *update* can be defined similarly as before.

```
lookupEnv :: Ref a env → Env t s env → t a s
lookupEnv Zero (Cons p _) = p
lookupEnv (Suc r) (Cons _ ps) = lookupEnv r ps
updateEnv :: (t a s → t a s) → Ref a env → Env t s env → Env t s env
updateEnv f Zero (Cons ta rs) = Cons (f ta) rs
updateEnv f (Suc r) (Cons x rs) = Cons x (updateEnv f r rs)
```

Sadly I could not get an updated version of the evaluator working because of infinite type errors.

```

eval4 :: Expr a (Env t use def) -> (Env t use def) -> a
-- eval4 (Var r) e = eval4 (lookupEnv r e) e
eval4 (IntVal      n) _ = n
eval4 (BoolVal    b) _ = b
eval4 (Add        e1 e2) e = eval4 e1 e + eval4 e2 e
eval4 (LessThan   e1 e2) e = eval4 e1 e < eval4 e2 e
eval4 (If e1      e2 e3) e = if eval4 e1 e then eval4 e2 e else eval4 e3 e

```

4 The transformations

The library defines the *Trafo* datatype that represents typed transformation steps and an *Arrow* interface like set of combinators for composing transformations. This uses some difficult existential datatypes, laziness on the type level and knowledge of the *Arrow* interface. Therefore I will only give a small taste of the transformation datatype and the complexity involved:

```

-- the datatypes for the transformations
newtype T e s = T { unT :: ∀ x. Ref x e -> Ref x s }
data Trafo m t a b = Trafo (∀ env1. m env1 -> m env1 -> TrafoE m t a b env1)
-- TrafoE is used as an inner datatype to be able to use existential quantification
data TrafoE m t a b env1 =
  ∀ env2. TrafoE (m env2)
  (∀ s. a s -> T env2 s -> Env t s env1 ->
   (b s, T env1 s      , Env t s env2)
  )

```

5 CHRISTMASTREE

Changing Haskell's Read Implementation Such That by Manipulating Abstract Syntax Trees it Reads Expressions Efficiently by Viera et al. [2].

Here parsers (or parser combinators) are the initial EDSL. However bottom-up parser do not compose, and are thus can not be transformed. Top-down parsers do not compose efficiently for arbitrary grammars.

Now what?

Solution: We compose grammars.

So now we have an EDSL, grammars, which are then coded as typed ASTs. These ASTs can then be analysed and manipulated (transformed) on runtime, thus allowing an efficient implementation of **deriving** *Read*.

A teaser:

```

class Gram a where
  grammar :: DGrammar a
  gread :: Gram a => String -> a
  gread = (parse o compile o leftfactoring
          o leftcorner o group) grammar

```

DGrammar is a data type that uses the *Env* and *Ref* as defined before.

group: Combines pieces of grammar and introduces extra non-terminals in the grammar to store precedences.

leftcorner: Remove left-recursion by a left-corner transform.

leftfactoring: Combine prefixes of alternatives (by left factoring).

compile: Map resulting Grammar onto a parser.

parse: Do the actual parsing.

References

- [1] Arthur Baars, S. Doaitse Swierstra, and Marcos Viera. Typed transformations of typed abstract syntax. In *TLDI '09: fourth ACM SIGPLAN Workshop on Types in Language Design and Implementation*, New York, NY, USA, 2009. ACM.
- [2] Marcos Viera, S. Doaitse Swierstra, and Eelco Lempink. Haskell, do you read me?: constructing and composing efficient top-down parsers at runtime. In *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*, pages 63–74, New York, NY, USA, 2008. ACM.